Criterion C: Development

Libraries imported

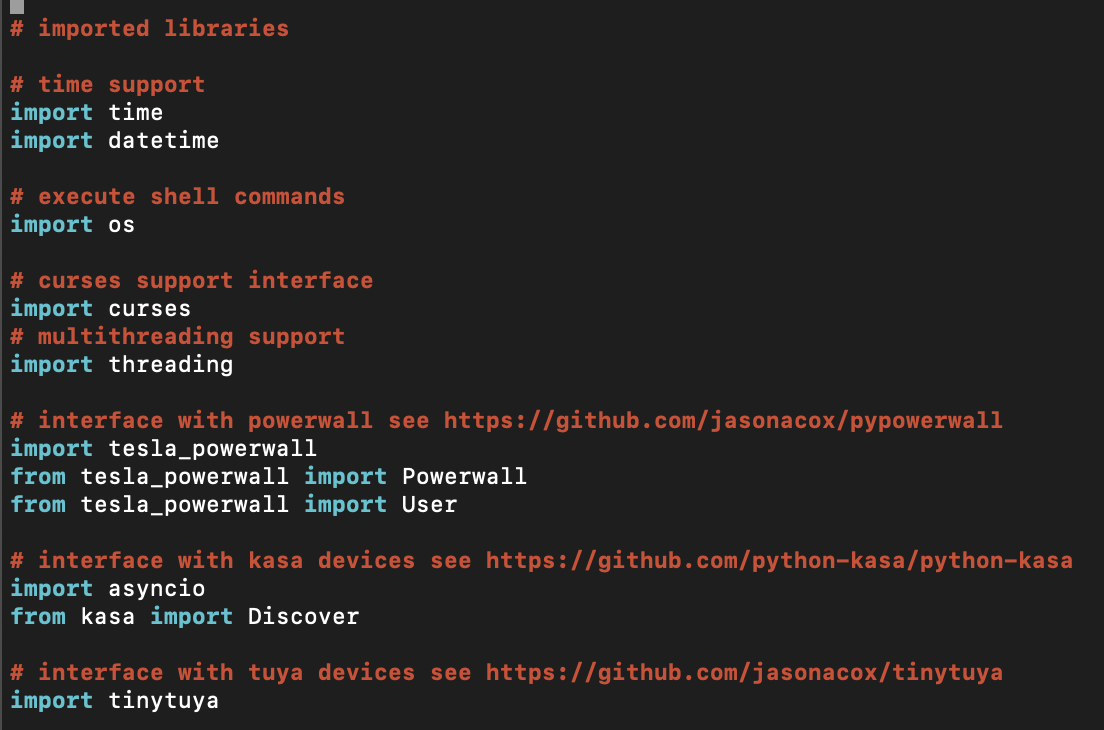


Figure C.1: Imported Libraries

The above system libraries are used in the program. Of particular note are the following:

* os: used to execute shell commands such as shutdown remote desktops and servers
* threading:is used to create separate threads for monitoring and user interface
* curses: allows for creation of a textual user interface in the terminal

The three libraries specific to the program functionality are: tesla\_powerwall library, kasa library and tinytuya. These allow the program to interface with the tesla powerwall, kasa devices, and tuya devices.

Both the tesla\_powerwall and kasa libraries were simple to set up.

The tinytuya had more complicated setup. It requires creating a tuya developer account on iot.tuya.com and then a cloud project under the account. The cloud project was named wallcontrol.

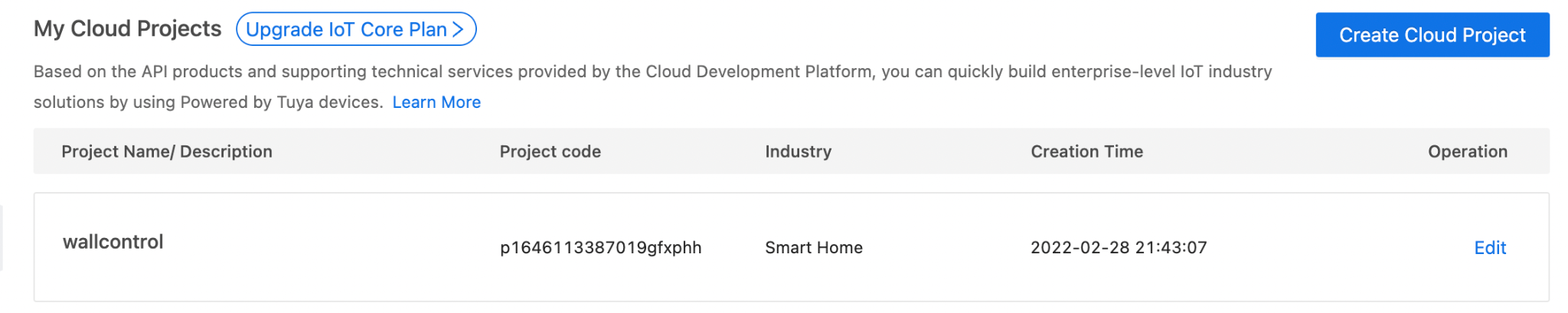


Figure C.2: Tuya Cloud Project

Then the client’s tuya mobile app account was associated with the cloud project to allow the cloud project to control the client’s devices.

Configuration Variables

After importing libraries, configuration variables are initialized. These variables have different values depending on the client. The list of variables includes: the path to the directory where the configuration files for the program are stored, the powerwall name and password, an email to send notifications to, and the tuya cloud account information. Configuration variables allow easy installation of the program for other clients.



Figure C.3: Configuration Variables (powerwall password and tuya cloud account secret hidden for security reasons)

Timing Variables

The timing variables that control how often the powerwall is probed for grid status, and how long the program waits before shutdown or bringup are initiated.

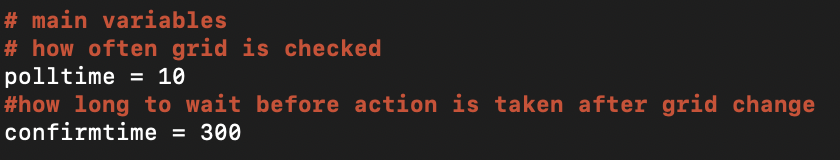


Figure C.4: Main Variables

Debug Control and Powerall Library Initialization

The debug variable controls generation of additional messages for debugging. Additionally, the powerwall library is initialized.

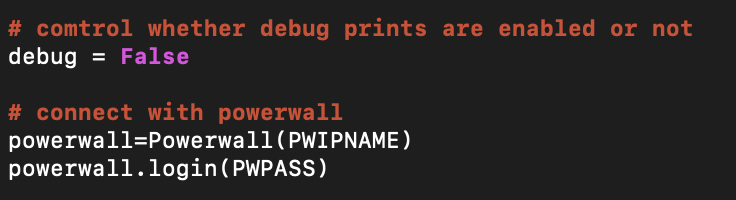


Figure C.5: Debug Variable and Powerwall Library Initialization

Main Body

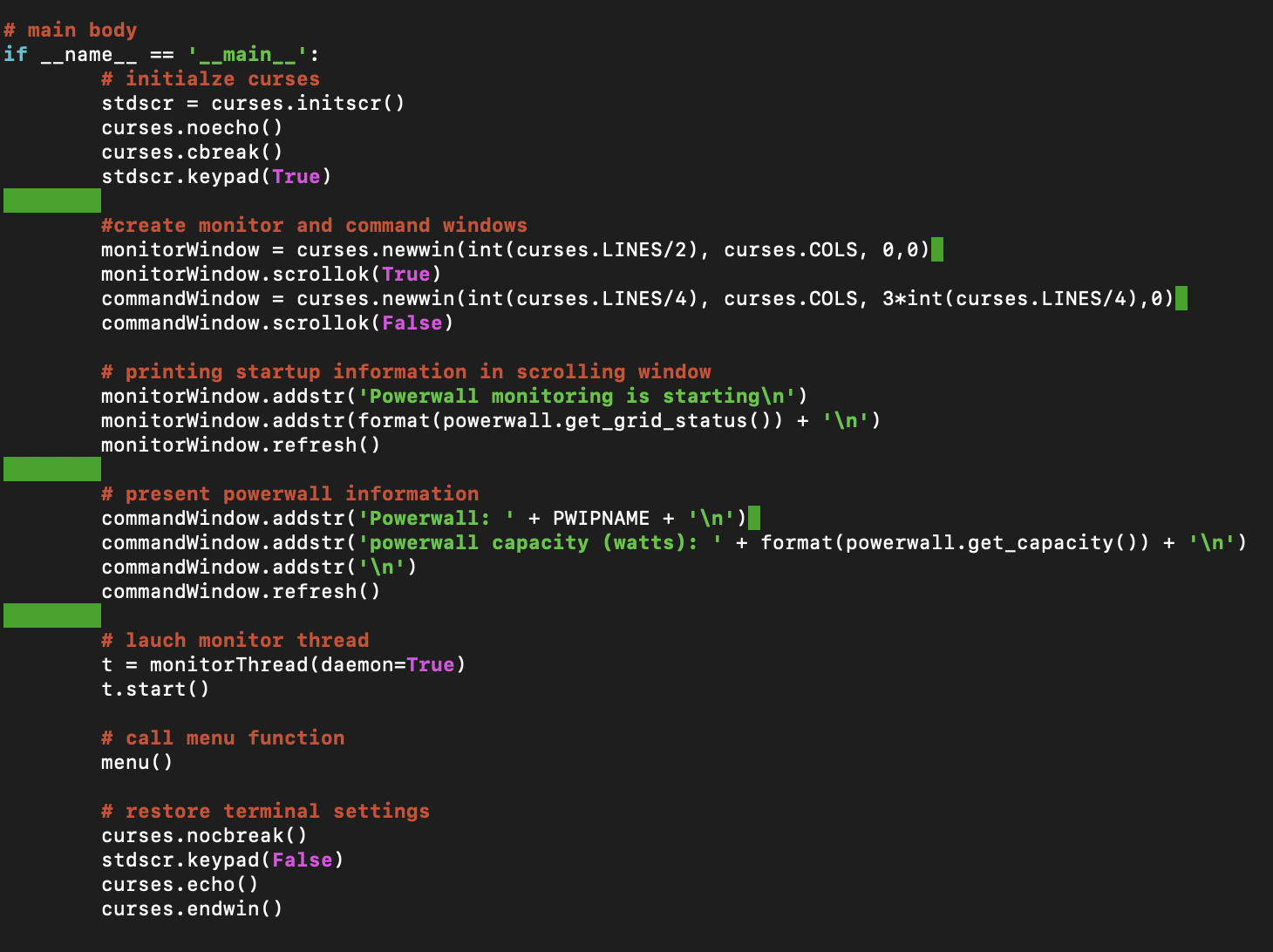


Figure C.6: Main Body

The section above does the following tasks:

* Initialize the curses library
* Create scrolling and command windows in the terminal
* Print startup information in scrolling window and powerwall information in command window
* Launch the monitor thread. The parameter *daemon* is set to True so that it will automatically terminate when the main thread terminates.
* Call the menu function from the main thread where it waits and processes user commands
* Upon exit, restore the default terminal settings.

Textual User-Interface

The image below is what the terminal looks like. The windows will be constructed using the curses library. The top half of the terminal will be a scrolling window that displays grid status and power use every minute. Additionally, it shows informative messages when devices are shutdown and brought up. The bottom third of the screen contains the command window for the user. There are three options displayed. 1. Test shutdown, 2. Test bring up and 3. Exit program. This allows the user to test the functionality without bringing the powerwall up and down. The user can press 1 2 or 3 and initiate the appropriate action. Additionally basic information (powerwall name, capacity and charge percentage) about the powerwall is presented in this window. The buffer area in between is there because sometimes libraries can print error messages without using the curses functions. This way it will not interfere with the menu.

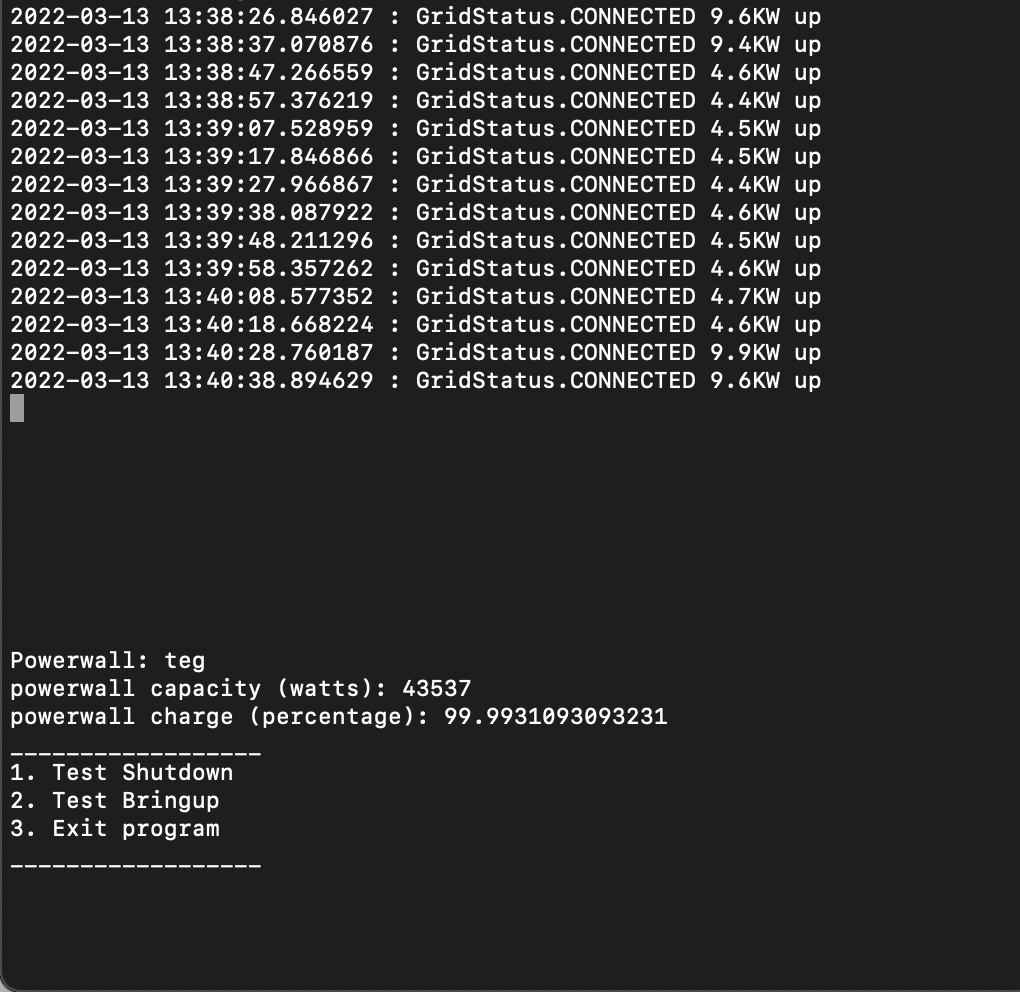


Figure C.7: Textual User Interface

Multithreading

Multithreading is used to separate the polling functionality from the user interaction functionality. The following snapshot shows the subclass used to launch the monitor function. The monitorThread class is made to be a subclass of the imported threading.Thread class. This is created for simplicity’s sake, so that the monitor thread can proceed while the main thread is waiting for user input. The main thread is executing a simple input loop.

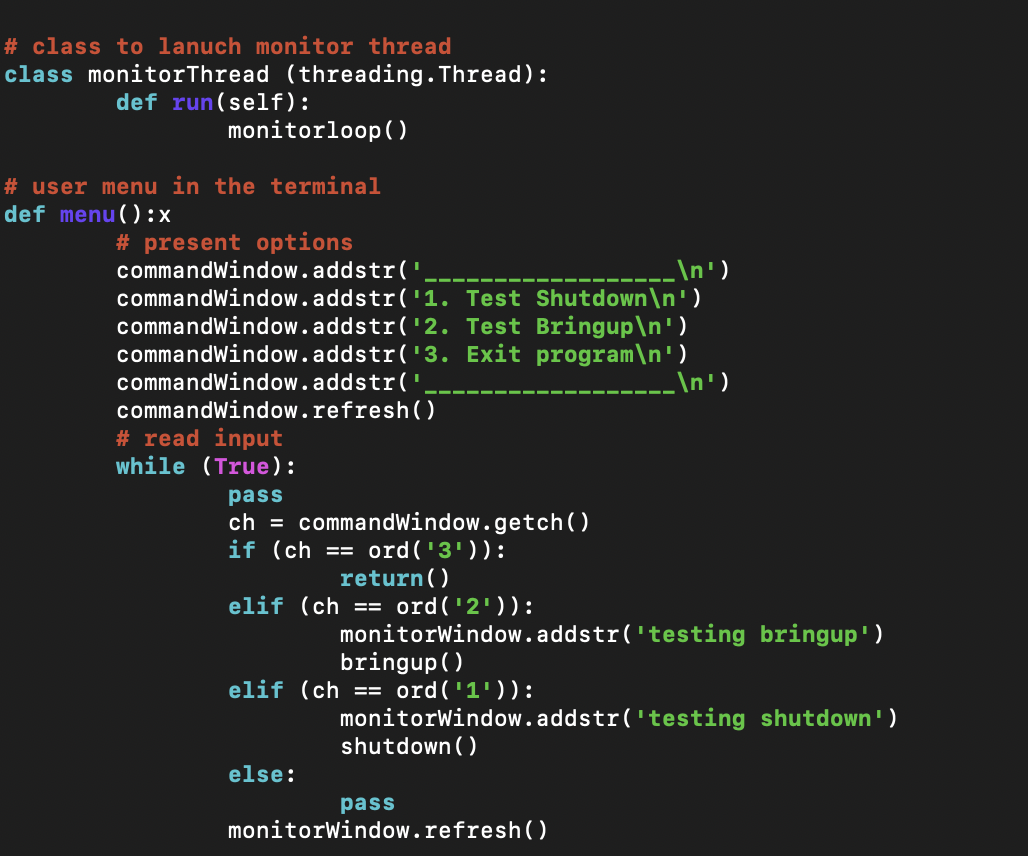


Figure C.8: Multithreading and menu function

Main Polling and Control Loop

This is the most critical part of the program that implements the desired load shed functionality. The *monitorloop()* function is called by the monitor thread. The outer loop is executing continuously until the program ends. As long as the grid stays up the program remains in the first internal loop. If the grid goes down, the *else* clause in the first loop is executed. It will first wait for *confirmtime* seconds before actually doing anything. If the grid is still down after this time, the shutdown is performed and the *down* variable is set to True. If the shutdown occurs, the thread moves to the second internal while loop, waiting for the grid to come up. The thread shutdowns tuya and kasa devices in every iteration because such devices are also controlled by other applications that the client has installed and they are automatically scheduled to come up at certain times in the day. If the grid comes up, the *else* clause in the second loop is executed. It will first wait for *confirmtime* seconds before actually doing anything. If the grid is still up after this time, the bringup is performed and the *down* variable is set to False. If not, the thread will proceed with the next iteration in the outer loop. Note that in this case the thread will attempt to run shutdown again, which is acceptable since the devices are already down.

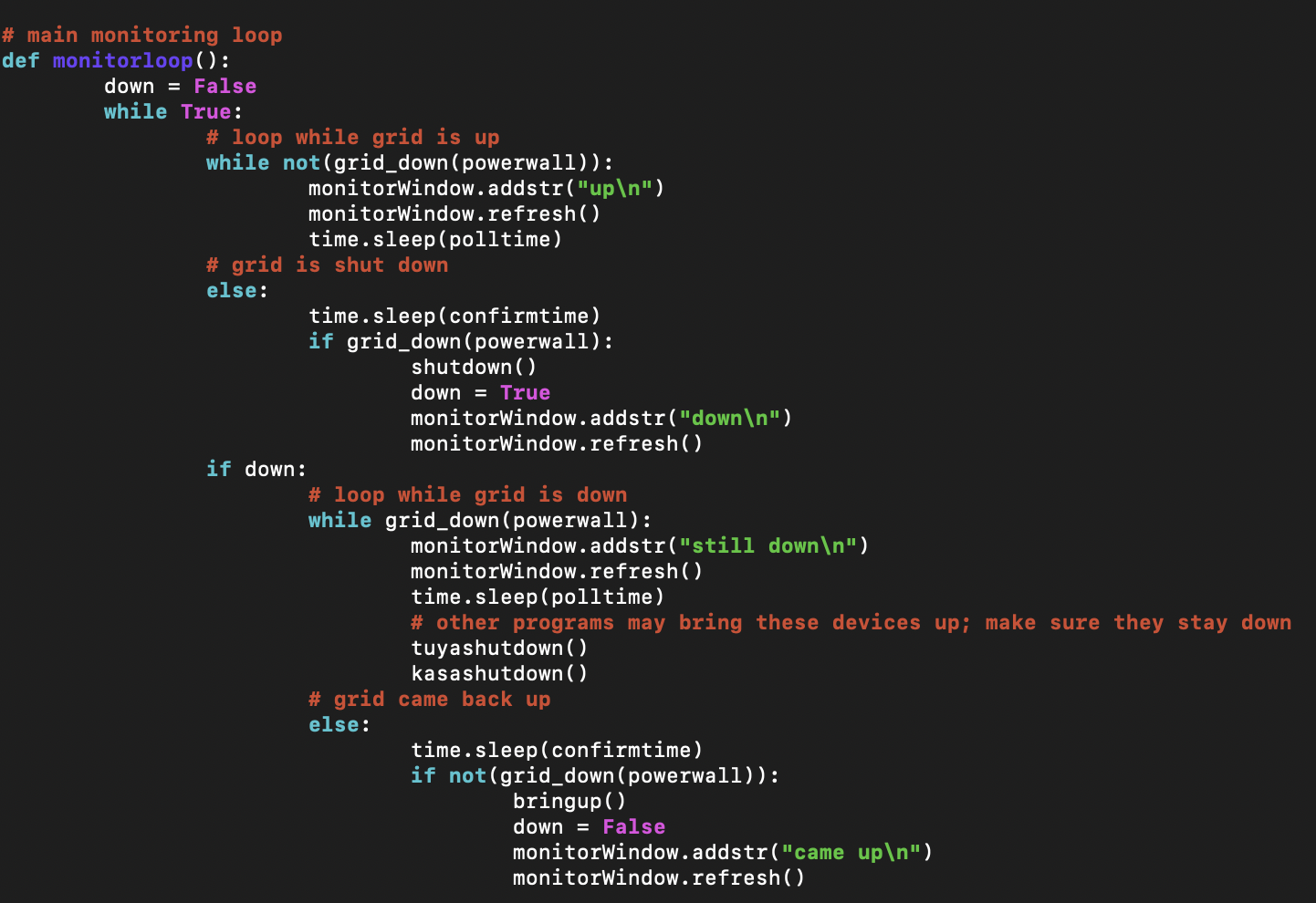


Figure C.9: Main Polling and Control Loop

Grid Polling Function

The grid polling function below checks the status of the grid, displays updated information in the command window, and returns a boolean value to indicate whether the grid is up or down.



Figure C.10: Polling Function

Main Shutdown and Bringup Functions

The shutdown function shuts down windows desktops, servers, kasa devices, and tuya devices. The bring up function brings up kasa and tuya devices. Bringing up computers was explored using WakeOnLan but did not work for computers connected wirelessly. Instead, the recommendation to the client was to use kasa or tuya devices to bring them up. Most computers can be configured to automatically come up when power is restored. Shutting them down programmatically is still vital so that data is not lost. Finally, an email notifications is also sent to the client.

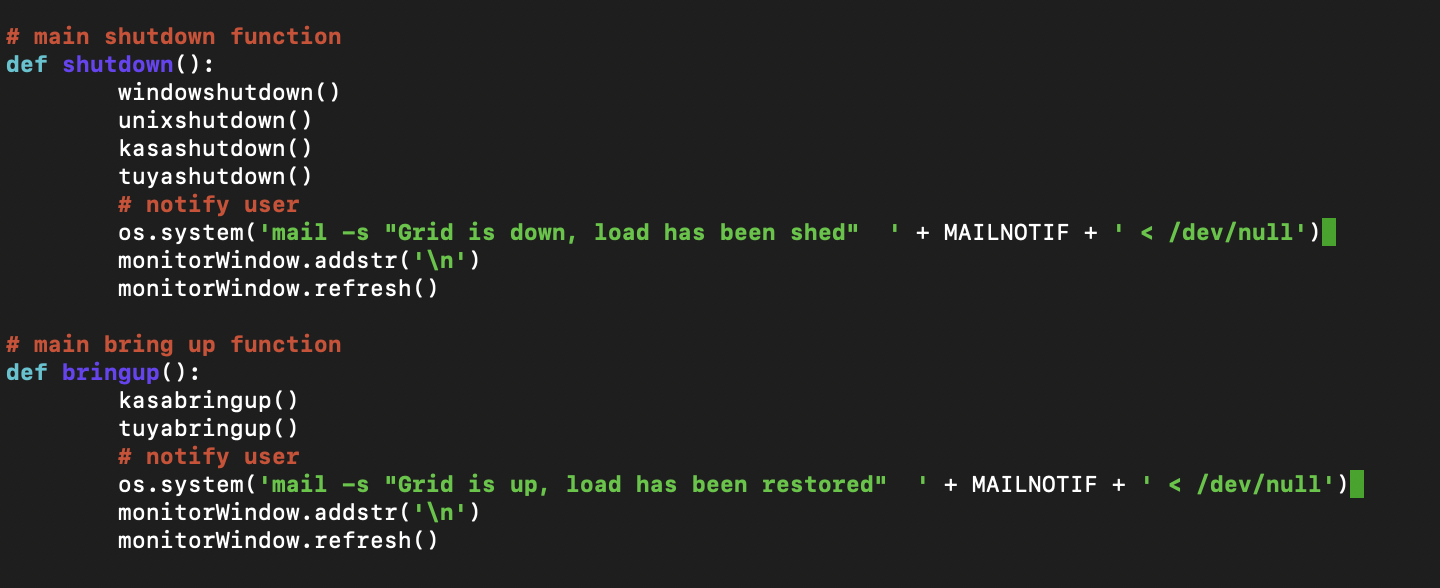


Figure C.11: Shutdown and Bringup

Windows desktop Shutdown

This function shuts down windows computers.



Figure C.12: Windows Shutdown

A text file is accessed for necessary information. This information is then used to remotely shut down the windows computers.



C.13: *windows.txt* Format

Each computer must be prepared according to specific instructions so that it can be shut down remotely. First, a security policy is added that enables remote shutdown. Second, a registry key is added to disable UAC remote restrictions (no user prompt required to shut down computers remotely). Finally, the remote registry service is started to enable administrative commands from remote computers.

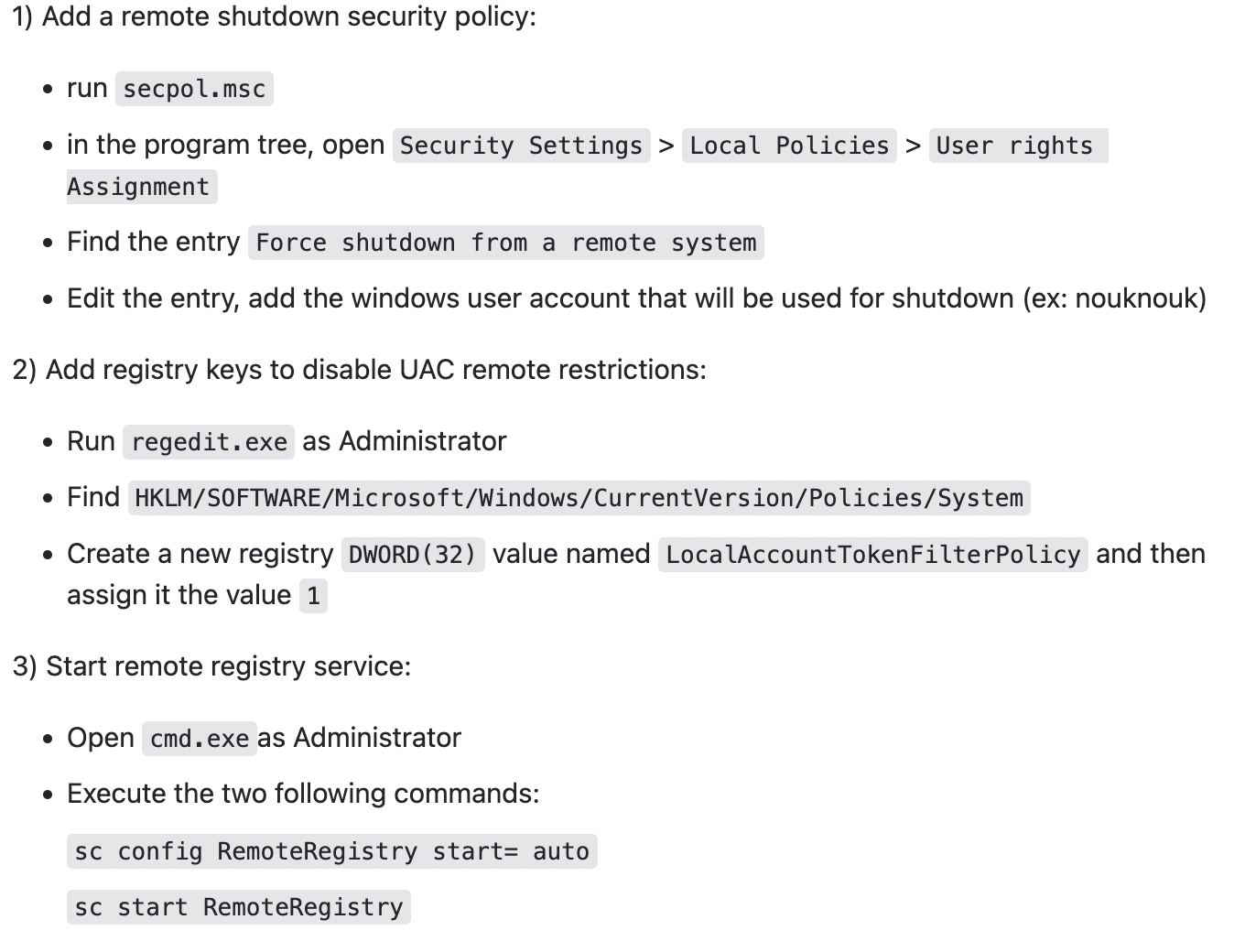


Figure C.14: Detailed Setup Of Windows Computer For Remote Shutdown

Samba was also installed on the monitoring Linux computer to provide the net command used by the program.

Server Shutdown

The *unixshutdown* function below shuts down servers. A text file is accessed for necessary information. This information is then used to remotely shut down the servers. The computer names from the servers.txt file. Some of the servers are linux opens and others are solaris. Since they require arguments for the shutdown command, the distinction is made within the text file.

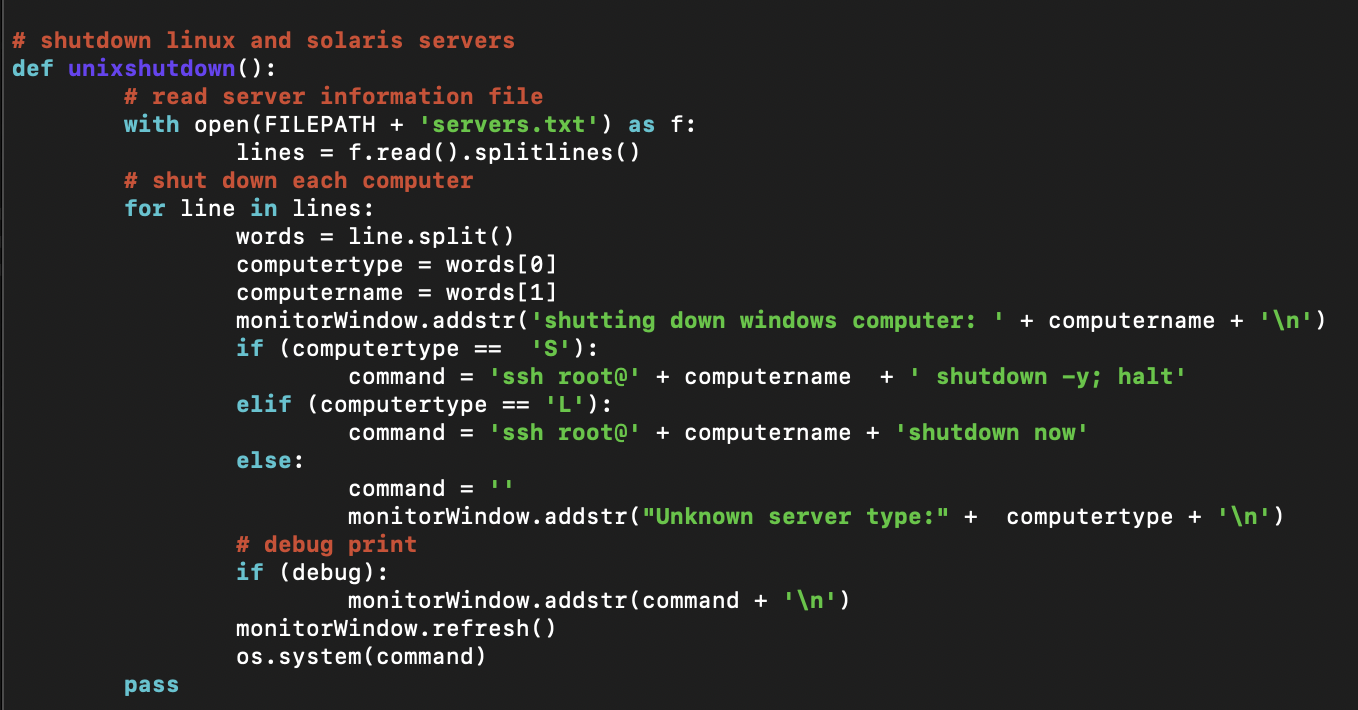


Figure C.15: Server Shutdown



Figure C.16: *servers.txt* Format

To prepare the servers, so that it is not necessary to provide username and password the following was done:

* On the monitoring computer the *ssh-keygen* command is run to generate ssh authentication keys
* For each target computer, the authentication keys are copied using the *ssh-copyid* command

Kasa Shutdown and Bringup

First, the devices are discovered and then each device is shutdown/broughtup.

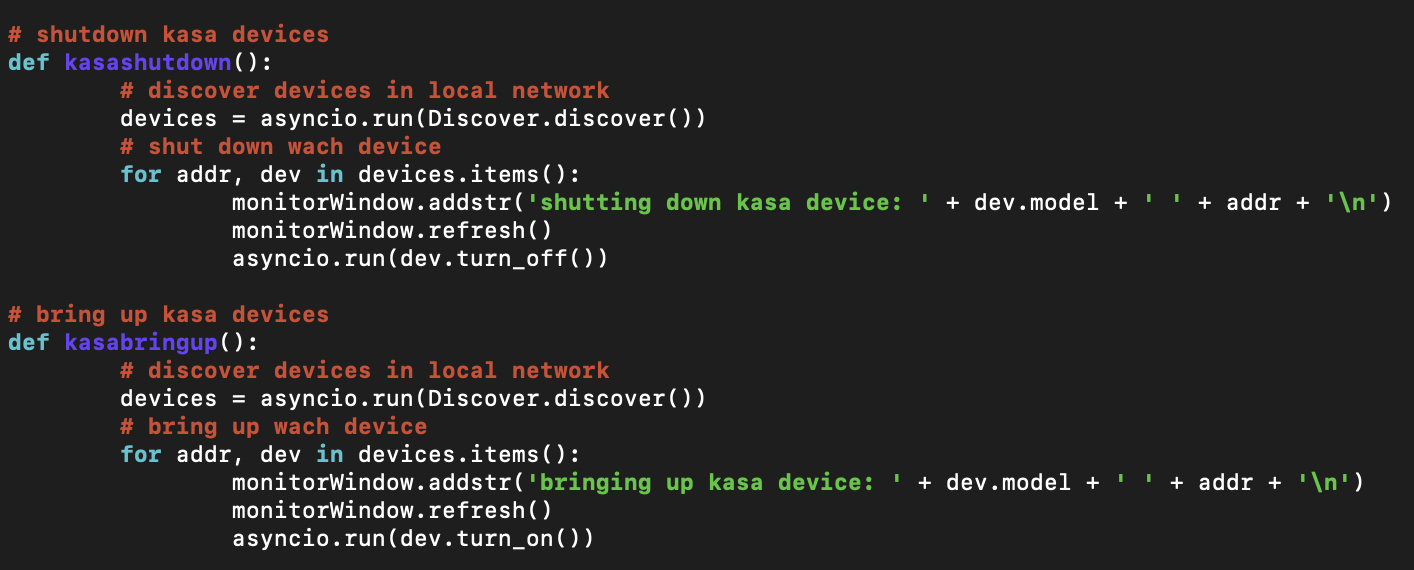


Figure C.17: Kasa Shutdown & Bringup

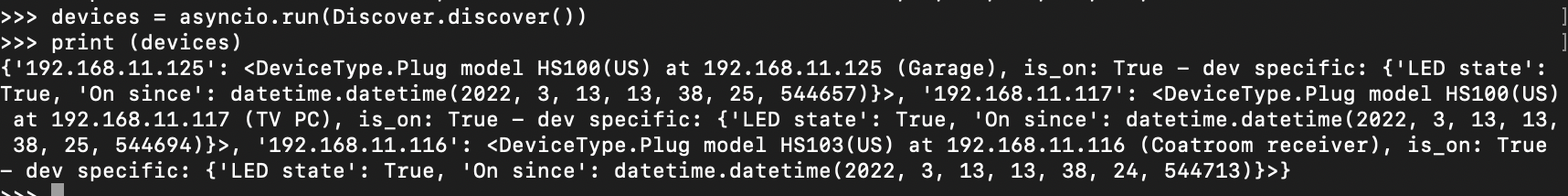
The discover function returns a structure that contains a record for each device:

Figure C.18: Discovered Kasa Devices

The for loop in the function iterates to retrieve each device record and then invokes the turn\_on/turn\_off function for the device..

Tuya Shutdown and Bringup

First these functions connect to the tuya cloud to get the list of tuya devices in the client’s home. Then, each device is shutdown or brought up. Because there are multiple types of tuya devices, both a bultshutdwon and switchshudown commands are sent to each device.

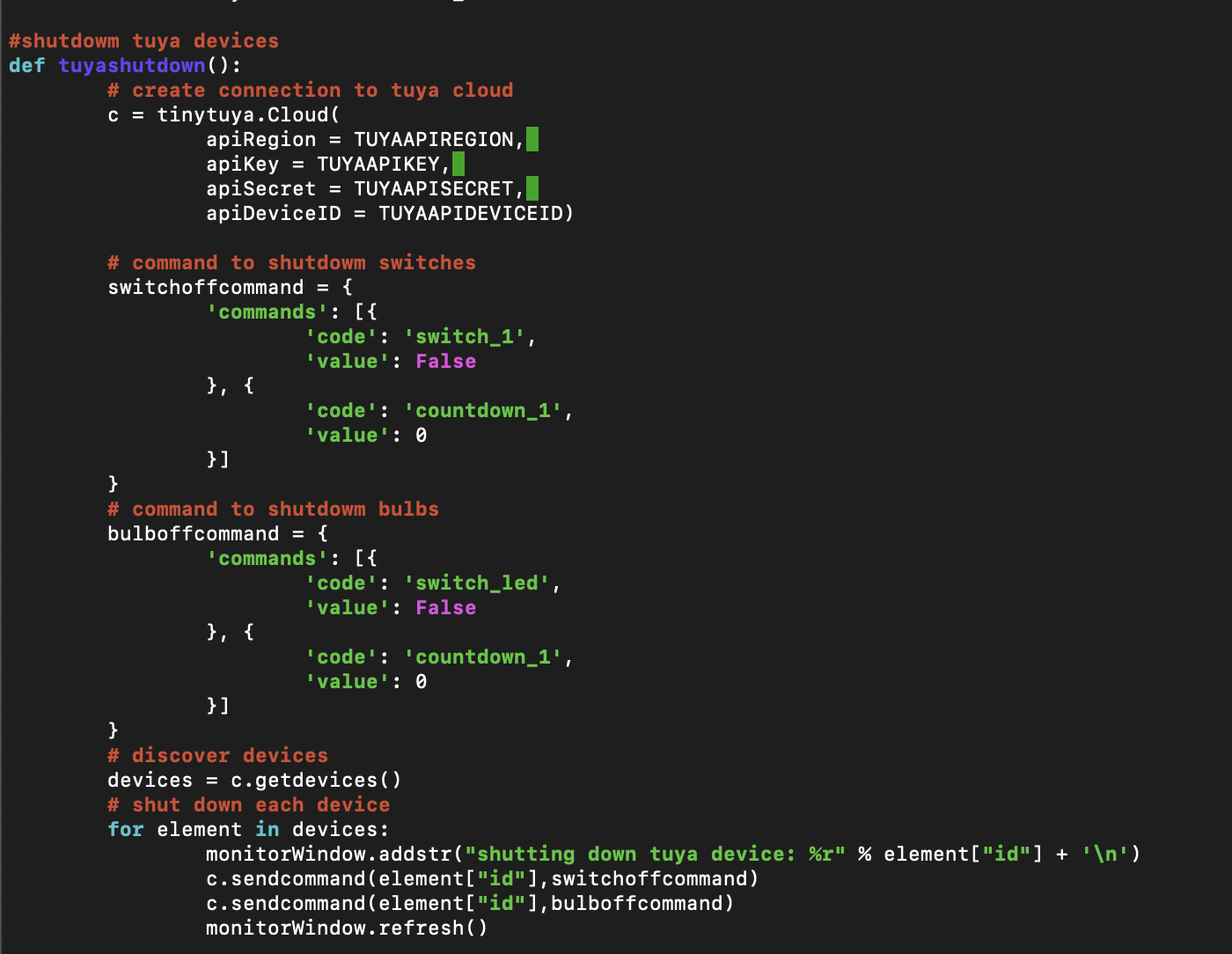


Figure C.19: Tuya Shutdown



Figure C.21: Tuya Bringup

The *getdevices()* function returns a structure that contains a record for each device. In order to turn off the device the device *id* is needed. That is retrieved by indexing each record using the keyword ‘id’.



Figure C.20: Tuya Devices Found